Engineering A Cache
Oblivious Sorting Algorithm
Gerth Brodal, Rolf Fagerberg and Kristoffer Vinther

Presenter: Rawn Henry
February 25 2019
Memory Hierarchy

Typical cache sizes:

L1 Cache: 32kB – 64kB
L2 Cache: 256kB – 512kB
L3 Cache: 8MB – 32 MB
Main Memory: 4GB – 32 GB
Disk: Terabytes
Observations

- Memory accesses are usually the bottleneck in algorithms since CPU is a lot faster than main memory.

- Want to minimize the number of times we have to get data from slow memory by maximizing data reuse.
Cache Aware vs Cache Oblivious

- Both have cache friendly access patterns.
- Cache aware algorithms depend on the parameters of the architecture such as cache size and depth of hierarchy whereas cache oblivious algorithms do not.
- Cache aware algorithms tend to be faster but are not portable without retuning.
- We want speed of cache friendly access but portability of cache oblivious algorithm.
Funnel Sort Algorithm Description

- Recursively sort $n^{\frac{1}{3}}$ contiguous arrays of $n^{\frac{2}{3}}$ items
- Merge the sorted sequences using a $n^{\frac{1}{3}}$-merger
- Base case is a merger with $k = 2$
- Similar to merge sort but different merging routine
Funnel Sort Picture

Taken from 6.172 lecture 15 Fall 2018
Sorting bounds

Quick sort
- Work = $O(n \log n)$
- Cache usage = $O(n/B) \log n$)

Funnel sort
- Work = $O(n \log n)$
- Cache = $O((n/B) \log_M n)$
Issues with Funnel Sort

- In practice it is not always possible to split K-Funnel into VK bottom funnels, it may lead to rounding errors.
- vEB layout performs well for binary trees but does not perform well for complex data structures.
Lazy Funnel Sort

- To overcome rounding problem, we use binary mergers
  - Takes two sorted streams and delivers an output of 2 sorted streams

- vEB layout is very friendly to binary trees

- Analysis of algorithm remains the same despite changes
Lazy k-Funnel Sort Diagram

Procedure

\[ \text{Fill}(v) \]

\[ \begin{align*}
\text{while out-buffer not full} \\
\text{if left in-buffer empty} \\
\quad \text{Fill(left child)} \\
\text{if right in-buffer empty} \\
\quad \text{Fill(right child)} \\
\text{perform one merge step}
\end{align*} \]
Algorithm Parameters

- Lazy funnel sort recursively sorts $N^{(1/d)}$ segments of size $N^{(1-1/d)}$ then performs a $N^{(1/d)}$ merge
- $\alpha$ – controls the buffer size
Optimizations: k-Merger Structure

- Memory layout
  - BFS, DFS, vEB
  - Nodes and buffers separate/together

- Tree navigation method
  - Pointers, address calculations

- Styles for invocation
  - Recursive, iterative
Optimal k-Merger Structure

- Swept k in [15, 270] and performed \((20 \times 10^6 / k^3)\) merges
- On 3 architectures found best configuration for merge structure was:
  - Recursive invocation
  - Pointer-based navigation
  - vEB layout
  - Nodes and buffers separate
Optimizations: Choosing the right merger

- Minimum of elements left in each input buffer and the space remaining in output buffer
- Optimal merging algorithm
- Hybrid of optimal merging algorithm and heuristic
- Simple

Simple was the fastest probably due to hardware branch predictions
Optimization: Degree of Merges

- Simple merge by comparing first elements
- Tournament trees

- Increasing merge degree decreases height of the tree meaning less tree traversals and data movement down the tree.
Tournament trees

Taken from 6.172 lecture 15 Fall 2018
Optimal Merge pattern

- Found 4 or 5 way mergers were optimal. Tournament trees have too large of an overhead to be worthwhile.
Optimization: Caching Mergers

- Each of the calls of the outer recursion use the same size k-merger. Therefore, instead of remaking the merger, it was simply reused for each recursion.
- Achieved speedups of 3-5% on all architectures
Other Optimizations

- Sorting Algorithm for base case:
  - GCC quick sort to avoid making mergers with height less than 2
  - Tuning parameters alpha (to control the buffer size) and d (to control the progression of the recursion)
Results

- Performance depended on architecture
  - Quick sort was better for architectures with very fast memory buses or slower CPUs so memory was not as much of a bottleneck
  - Funnel sort generally outperformed quicksort for larger n
Results

Uniform pairs - AMD Athlon

Walltime/n\cdot \log n

log n
Results – Faster Memory Arch
Results – QS Memory Sensitivity

![Graph showing memory sensitivity for QS algorithm with different types of sorting.](image-url)
Results – External Sorting